The OSF DCE Developers Conference

DCE Success Stories Track

DCE and Legacy Systems -An Experience Report What Really Happened

By: John Diehl, with Randy Parlier and Timothy Graham
California Institute of Technology
Jet Propulsion Laboratory

Abstract

The Multimission Ground Data System (MGDS) in use at the Jet Propulsion Laboratory (JPL) was developed in the latter half of the 1980s. It was a major departure from the one-of-a-kind, non-distributed ground data system previous] y employed. Today, a project is underway to determine if the Distributed Computing Environment (DCE) has a place in MGDS. The initial component targeted for replacement is an application layer built on top of TCP/IP which handles the MGDS message-passing requirements.

This paper is intended to:

- Share our experience with other DCE developers
- Help other developers understand how DCE can be used "in the real world
- Share some of the limitations of DCE with other developers

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

Table Of Contents

OVERVIEW

In the paper published in the conference proceedings, 1 presented several areas within the Multi-mission Ground Data System (MGDS) which arc candidates for replacement with DCE-based solutions. Since completing the paper, 1 have concentrated on constructing a DCE solution to replace the Network Access Layer of MGDS. This presentation will focus on my experience creating a prototype of the replacement.

I will first present the lessons learned from building this prototype. Next, I will describe the application which I hope to replace, and the prototype. Finally, 1 will give an evaluation of DCE as a vehicle for replacing legacy systems.

My ccl] contained the following workstations:

1 Sun Sparcstation 10 .
1 Sun4 Spare
1 Hewlett-Packard 725/50

LESSONS LEARNED

Did I Ever!

I have been involved in data processing since the early 1970s. In 1972 Burroughs Corporation hired me to perform operating systems' maintenance. Since then I have been working in the data processing field, and have seen many innovations flourish and fold. 1 have learned many lessons the hard way. Still, all this experience did not protect mc from learning, yet again, several more lessons the hard way. Simultaneously I think my experience did allow mc to do some right things the right way.

2.1 Reading

1 feel compelled to parrot the words of the knowledgeable people with whom I discussed DCE:

"Read the two O'Reilly books: *Understanding DCE, by* Ward Rosenberry, David Kenncy and Gerry Fisher, and *Guide to Writing DC Applications*, by John Shirley."

To which I add: after you read them, do the example applications,

DCE is a complex system. You need all the help you can get to understand the services offered and how to use the services to obtain results. These two books can provide a firm base from which to explore DCE. Even now, I return to these books to find out about a "new" idea or capability that I have just discovered.

A new book published by McGrawHill which I am just now beginning to use is *OSF DCE Guide to Developing Distributed Applications,* by Harold W. Lockhart,

The first two books are an "easy" read. The last book is larger, and I am finding it useful for learning more about the topics introduced in the O'Reilly books.

2.2 Training

This is onc thing I mostly did right. Before undertaking to set up anything using DCE, I would recommend getting as much training as possible. 1 took a three-day class from TRANSARC and attended a series of Tutorials/Workshops offered at the SHARE conference in Anaheim, California. The benefits of training would have been even greater if 1 had even more training, especially of the hands-on variety. TRANSARC's class was lecture only. The SHARE conference included a two-hour hands-on session which was extremely useful. Hewlett-Packard offers a one-week hands-on class which, because of my own schedule, I was unable to attend. I ruc that schedule conflict, In retrospect, 1 would have benefitted by traveling to the class rather than waiting for it to be offered locally.

1 spread my training over two or three months. When I was not in training classes, I speculated about how my replacement system might look. I did not start work on it until after the paper was submitted early in July. During this time I was busy collecting my sample applications and reading.

2.3 Sample applications

Work the sample application in the texts. I found that differences in the vendor products made the applications as presented lcss than robust, 1 am not a true UNIX programmer, so I found the incompatibilities in the Makefiles especially trying. However, once 1 had done the examples and had prevailed in creating workable Makefiles, I reduced the challenge of creating my own application. It would have been far more difficult to create a working application had I needed to cover all the ground on a new, untested piece of software.

The Hewlett-Packard DCE package included a set of thirteen (13) sample applications. These came with documentation which was very helpful in learning about DCE. HP heavily documented their sample application code; this was a real boon when I was building my prototype application. Of course the O'Reilly sample applications come with the book, and are equally easy to understand.

As time goes by, 1 am adding to the sample application list. The application I will discuss later is now part of my sample application suite.

2.3.1 As a training tool

About six months after I began working on DCE, a different organization hired two people to explore DCE. As part of my outreach program at JPL, 1 helped bring these individuals up to speed. They began by reading the O'Reilly books; they then worked the sample application provided in these books and by Hewlett-Packard. They completed this training and built two working application in less than three (3) months. I believe the sample applications were a key ingredient of their success.

I would point out that not all the sample applications work. When these two individuals joined the team, about six (6) of the applications worked. Now, of the 20 sample applications I have, 15 work on both platforms and 18 work on at least one platform.

2.3.2 As a development tool

As I said, HP heavily documented their sample applications. As 1 was developing my applications, I found it very useful to find a similar application in the samples. I would usc these samples as the basis for constructing my "new" application. My prototype application was too unique for plagiarism, but I still used the sample applications as sources for information on which calling sequences to usc.

Having these sample applications saved many hours of work.

2.4 Usc available people resource

Others who are knowledgeable about DCE arc an excellent resource; even individuals who know only the barest minimum about DCE -- they have read the books -- can understand what you arc trying to do, They can provide insight into your own problems and help you reach your goal more quickly.

People with minimum knowledge who merely listened to mc were very useful. 1 also relied heavily on the two people, mentioned above, who worked in the other organization. Wc provided mutual support.

2.5 Multi-vendor environment

Another advantage I had in developing my application was a multi-vendor environment. This hclpcd separate system problems from application problems.

When I had a failure on one vendor's implementation, I would take the application to the other vendor's platform and try it thcrc. Each implementation had its strengths and weakness. Often a "core file" on one system resulted in an error message on the other. The error message usually led to a discovery of parameter errors which would have been difficult to find manually. I also used my sample applications to decide if the call I was using was correct.

As onc of my sponsors aptly put it: "Sometimes value added is not always valuable."

Another advantage was the fact that HP had implemented the cdsbrowser while TRANSARC had not. I began by using the cdscp and rpccp commands to monitor my application. The cdsbrowser man page states that the capabilities in the browser arc available using cdscp. However, there was a wealth of information about the servers which I was unable access using these programs. The same information was ready to hand using the cdsbrowser. The cdsbrowser would have been lcss useful outside a "DCE" environment. In a ccl] it dots not matter which node you are on, so I could run the cdsbrowser on the HP node and view the CDS on the Sun server. Excellent!

2.5.1 Location of vendor-specific errors

The only error I succeeded in isolating and repeating was in the rpc_string_binding_parse. The documentation states that you may specify NULL as an address parameter and the routine will not return the value. Not in the HP and TRANSARC implementations. Both versions produce a core file when this is attempted. Eventually, 1 simply requested all the values and ignored the ones I did not need.

2.5.2 Verification of implementation

As 1 mentioned, I used the multi-vendor environment to test-verify the portability of the implementation. The portability of the application aided in timely completion of the prototype.

2.6 Problem description

1 spent a great deal of analysis time trying to describe my application to other people. This was necessary as I worked through my failures, but was fairly frustrating since everyone seems to have their own definitions of network-related terms. It was not until I stumbled across a familiar paradigm to describe the problem that I was able to solve it.

My advice: if you find yourself (as I did) spending too much time describing the problem to others, spend some time trying to come up with a more familiar example of what you are trying to do.

3 DATA TRANSPORT SUBSYSTEM (ICE CREAM)

So, what was I trying to prototype?

The Data Transport Subsystem (DTS) provides Network services to the applications within MGDS. The specific service I targeted was the ability to deliver a stream of data from one application to another. DTS provides this service by first having the originating application "open" a DTS virtual circuit. When the destination application opens the same virtual circuit, DTS establishes a connection and the originating application "sends" data to the destination application which "reads" the incoming data. Of course this is a bare-bones description, but it is about as complex as 1 want to get here. Besides the application software, DTS makes use of a Logical Name Server (LNS) which is used by DTS to resolve the name of the virtual circuit to an endpoint. I should also point out that the life span of a virtual circuit is finite, Once a virtual circuit is created, it can last for anywhere from minutes to hours.

I soon pictured each virtual circuit as a service, but quickly realized that DCE was not built to allow a large volume of servers to be created and destroyed on the fly. So, 1 decided to use IDL to describe the DTS interface. 1 have represented the DTS as delivering a stream of data. In actuality, the stream is made up of component parts called standard formatted data units (sfdus). These are easy to describe in IDL and so the description of the interface is easy.

In the parlance of the text books, I wanted to create a service that could be offered by many servers, and I wanted to allow my client to select the specific instance of the server it wanted. This led to a several-day task searching for a way to use object uuids in DCE. Now that I have done this, it is obvious, but the long and winding trail lead to frustration as I tried to describe my application to others. At this point I hit upon the idea of describing my application as Ice Cream.

Assume you arc Ben&Jerry's™. You have many icc cream flavors, but the interface to each is pretty much the same. There are characteristics of each flavor which make it unique, but all the flavors share some common traits. I decided to implement Icc Cream as a set of servers, where each server is running for a specific flavor. The attribute I chose to supply to my ice cream clients was the number of calories pcr serving.

To relate this to my application, each flavor was a virtual circuit. The number of calories pcr serving was the stream of data. I had earlier developed and tested an interface which did supply a stream of data using DCE, so 1 was not concerned about generali zing the Icc Crcarn prototype to the DTS prototype.

There are actually two applications which make Up the Ice Cream prototype. The Logical Name Service (LNS) application maintains a database of the flavors. LNS allows a server to register a flavor and returns an object uuid to the server. LNS also allows an Ice Cream client process to determine the object uuid which has been associated with a particular flavor. The second application is Ice Cream itself.

Appendix A contains a description of the LNS application. Appendix B contains the listings of the source and makefiles for the LNS application. Appendix C contains a description of the Ice Cream application. Appendix D contains listings of the source and makefiles for the Ice Cream application.

I discovered four things in the crunch to create this prototype in time for the conference:

1) You can use Object UUIDs to distinguish between different servers offering the same service.

2) You can include code in your application that will guarantee only one server is offering your specific service.

3) You can include code in your application that cleans up the CDS when your application stops.

4) You can include code in your application to verify that an entry in the CDS is a valid server.

Items 3) and 4) were important to me since my application can have services come and go quickly. Ensuring that the named service I was offering was not a duplicate was critical to ensuring that my DTS prototype would be accepted.

All the example progams are heavily instrumented with calls to puts and printf. This is probably overdone, but I found overdoing better than underdoing, in this case.

I would also like to point out that toward the end of my effort, I discovered that the LNS application may not be necessary. It is possible to use the rpc_ ns_binding_export to place object uuids in the CDS. I did not have sufficent time to verify that before the conference.

4 DCE

1am an easy sell; I believe what I read. DCE as a concept was hard to resist. So how do I feel some six months later?

4,1 I like DCE

I was impressed by the fact that once I started to work on programming my application, it took me only 80 hours to finish it. This included the time required to learn that trying to describe the actual application was not very productive. I learned that lesson when I changed to the Ice Cream paradigm.

4.2 DCE Is portable

Usually, if 1 could run an application on *one* type of hardware, I could run the same application anywhere in the cell. The exceptions arc the result lack of time to port the dreaded Makefiles. As I stated earlier this is not really difficult; it simply is not where my interest lies.

4.3 Ready for prime time

After three months on the job, 1 began using the line: “It will be reasonable to plan to deploy a DCE-based application during 1996.” By that I mean that DCE as a product will be more stable after it has more releases under its belt and more application programmers are available who have used it, if only in a prototype environment.

I pretty much stand behind that statement today. However, depending on the size of the application and the number of programmers you are planning to use, I think the time to start planning the 1996 deployment is probably past.

<<This Page Intentionally Left Blank>>

APPENDIX A
DESCRIPTION OF THE LNS APPLICATION

Introduction

This appendix describes the logical name server (lns) application. It discusses how to build and run it, and what administrative setup is required to run it. It is assumed that you already have a properly configured and running DCE on the node(s) on which you run this sample application.

Description

The LNS application provides a data base which relates logical names with an object uuid. The code segments in Appendix B contain the server code and several pieces of client code which can be used to exercise the LNS server.

The LNS server verifies that no other LNS server is running in the cell before it starts. If an entry exists in the CDS for the LNS server, the LNS server checks to see if the other server is running by using the rpc_mgmt_is_server_listening call. If the other server is running, the server terminates.

The LNS server includes signal-handling code. When the server intercepts a signal it will unregister its binding handles and endpoints before terminating. This makes it easier on the program when it is restarted.

Once it is running, the LNS server supports the following functions:

register_logical_name	creates an entry in the logical name table which relates the user supplied logical name with an object uuid which is generated by the LNS server. This function returns the object uuid to the caller.
resolve_logical_name	looks for the user-supplied logical name in the logical name table. If the logical name is found, the LNS server returns the corresponding object uuid. If the user-supplied name dots not exist in the logical name table, the LNS server returns a non-zero status and a nil object uuid.
delete_logical_name	looks for the user-supplied logical name in the logical name table. If the logical name is found, the LNS server clears the logical name table entry and returns a zero status code. If the logical name is not found, the LNS server returns a non-zero status code.
print_ logical_names	prints a list of the registered logical names on stdout out for the server. (This is largely useful in testing the server.)

Each of these server functions is coded in its own C file.

The samples include source code for the following client programs:

register_client	registers a logical name with the LNS server. Upon successful completion, the program prints the returned object uuid.
resolve_client	uses the LNS server to resolve a logical name to an object uuid. Upon successful completion, the program prints the object uuid associated with the logical name.
print_client	Causes the LNS server to print a list of the logical names and their associated object uuid on the server stdout.

Administrative Setup

The LNS server uses the environment variable RPC_DEFAULT_ENTRY to identify the directory in which it registers its endpoints.

Build Process

This sample comes with two makefiles: Makefile.hp and Makefile.sun. Makefile.hp should be used on HP platforms; it places the executables in the hp subdirectory. Makefile.sun should be used on Sun platforms; it places the executable in the sun subdirectory.

The flag -DTRACING is defined by default in the makefiles. Turning this flag off will greatly reduce the number of displays printed by the LNS server.

Running the server and clients

To start the server enter:

```
<MACHTYPE>/DCE_lns_server
```

Each of the supplied client programs can be started in the same way.

APPENDIX B
SOURCE AND MAKEFILES LISTINGS FOR THE LNS APPLICATION

DCE_lns.idl:

```
/* File Name: DCE_lns.idl                 */
/* Purpose:   Define the LNS interface idl. */
[
uuid(09735c0c-8d19-11cd-bc6f-080009786a45),
version(1.0)
]
interface DCE_lns
{
        const  unsigned32              max_name_length = 1024;

        typedef [string, ref] char        logical_name_t[max_name_length];

        uuid_t register_logical_name (
                [in]     logical_name_t         logical_name,
                [out]    error_status_t         *call_status
        );

        uuid_t resolve_logical_name (
                [in]     logical_name_t         logical_name,
             [out]       error_status_t         *call_ status
        );

        void print_logical_names ( );

        void delete_logical_name (
                [in]     logical_name_t         logical_name,
                [out]    error_status_t         *call_ status
        );
```

DCE_lns_server.c

```
/* FILE NAME: DCE_lns_server.c                                    */
/* Purpose: The DCE_lns_server provides the services necessary to create */
/*          logical names, delete logical names and to relate          */
/*          logical names to a specific DCE server.                   */

#ifdef TRANSARC
#include <sys/machsig.h>
#endif          /* TRANSARC */
#include <signal.h>
#include <pthread.h>              /* POSIX threads facility */
#include <unistd.h>               /* Standard POSIX defines */
#include <stdlib.h>               /* Standard POSIX defines */
#include <string.h>               /* str*() routines */
#include <stdio.h>
#include <dce/dce_error.h>
#include <dce/rpcexc.h>
#include <dce/uuid.h>
#include "DCE_lns.h"
#include "check_status.h"

#define  MAX_LOGICAL_NAMES                  100
#define ANNOTATION_LENGTH                   050

/*
 * This server is a threaded process. To properly handle user-generated
 * (asynchronous) signals we spawn a new thread that will use the sigwait()
 * CMA routine to await the receipt of an asynchronous signal. When such a
 * signal comes in, the server is made to shut down gracefully.
 */
pthread_addr_t          sigcatch(pthread_addr_t arg);

struct lnt_entry {
        logical_name_t          user_logical_name;
        uuid_t                  logical_name_object;
        int                     user_count;
} lnt_table_entry [MAX_LOGICAL_NAMES];
```

```
main ()
{

  char                    annotation [ANNOTATION_ LENGTH];    /* Annotation for server */
  rpc_binding_handle_t    binding_handle;        /* refers to another server */
  rpc_binding_vector_t    *binding_vector;       /*set of binding handles(rpcbase. h)*/
  unsigned_char_t         *entry_name;           /*entry name for name service (lbase.h)*/
  char                    *getenv();
  int                     i;
  rpc_ns_handle_t         import_context;        /* used to disambiguate the name server*/
  boolean32               m atch_found;          /* Used to locate duplicate servers */
  pthread_t               sig_thread;            /* thread Id of signal handler */
  unsigned32              status;                /* error status (nbase.h) */
  ndr_char *              string_ binding;       /* used to create binding */
  unsigned32              _ignore;               /* error status (nbase.h) */

/* initialize table. */
  for ( i = O; i <=  MAX_LOGICAL_NAMES;  i++)
  {
    strncpy ( (char *) &lnt_table_entry[i].user_logical_name, " ", 1 );
    uuid_create_nil (
              &lnt_table_entry[i].logical_name_ object,
              &status );
    lnt_table_entry[i].user_count = O;
  }
```

```
   /* pthread_create() --
    *
    * Create a new thread to perform asynchronous signal handling. The
    * pthread_create call spawns a new thread of execution within this
    * process. The first paramter is the address of' the thread information
    * data structure; pthread_ create will fill this in as a result
    * paramter. The second paramter is the attributes to be used when
    * creating the new thread; the defaults are fine in this case. The
    * third paramter is the name of the function to call when the new
    * thread has been created; when this function returns the thread will
    * terminate. The fourth paramter is used to pass information to the
    * thread routine; the sigcatch routine does not need any additional
    * information, so a NULL pointer is passed in.
    */
   if (pthread_create(&sig_thread, pthread_attr_.default, sigcatch, O) < O) {
       /*     perror() --
        *
        * Print an error message using the string passed in and the current
        * value of the global UNIX error value, errno. The pthread_create
        * call will set errno if it fails.
        */
       perror("Cannot start signal catching thread");
   } else{
       /*     pthread_yield() --
        *
        * Force a context switch from this thread to another. In this case
        * there is only the one other thread, the one just spawned. Yield
        * here to allow the signal thread to set itself up before resuming
        * with registration of the server,
        *
        * NOTE: This does not guarantee that the signal catching thread
        * will run until it blocks (until the sigwait() call, see below).
        */
       pthread_yield();
   }
#ifdef TRACING
  printf("Thread id = %i.\n", sig_thread);
#endif          /* TRACING */
```

```
    /* rpc_server_use_all_protseqs() --
     *
     * Specify that the RPC runtime should use all protocol sequences for
     * this application (both UDP/IP and TCP/IP are currently supported).
     * This allows the client the flexibility of choosing whichever protocol
     * sequence it prefers; it also uses more system resources on the server
     * but that's OK for purposes of demonstration.
     *
     * The first parameter specifics the maximum number of concurrent remote
     * procedure call requests that the server can accept, This server
     * wishes to allow only 1 call at a time. The second parameter is the DCE
     * error status.
     */

#ifdef TRACING
  puts("Requesting all protocol sequences (rpc_server_use_all_protseqs) . . . ");
#endif          /* TRACING */
  rpc_server_use_all_protseqs(                /* create binding information */
     1,                                /* queue size for calls= 1 */
     &status
  );
  CHECK_ STATUS(status, "Can't create binding information", ABORT);

   /* rpc_server_register_if
    *
    * Register the interface definition and manager entry point vector with
    * the RPC runtime. This application does not usc type UUIDs (an
    * advanced feature) so specify a nil manager type UUID.
    */
#ifdef TRACING
  puts(''Registering interface (rpc_server_register_if) . ..").
#endif          /* TRACING */
  rpc_server_register_if(      /* register interface with the RPC runtime */
     DCE_lns_v1_0_s_ifspec, /* interface specification (DCE_lns.h) */
     NULL,                  /* No type uuid */
     NULL,                  /* Usc default end point manager*/
     &status                /* error status */
  );
  CHECK_STATUS(status, "Can't register interface\n", ABORT);
  / *   rpc_server_inq_bindings
   *
   * Get the bindings handles. The binding information (binding
   * vector return argument binding_vector) is required for registration
   * with the endpoint mapper and the name service.
   */
#ifdef TRACING
  puts("Obtaining server binding information (rpc_server_inq_bindings) ... ");
#endif           /* TRACING */
  rpc_server_inq_bindings( /* obtain this server's binding information */
    &binding_vector,
    &status
  );
  CHECK_STATUS(status, "Can't get binding information", ABORT);

#ifdef TRACING
  /*
   * Print out the bindings obtained from the RPC runtime. This info is
   * only for debugging purposes -- it shows what protocol sequence and
   * ports have been grabbed by the runtime for this server.
   */
  puts("Bindings:\n");
  for (i = O; i < binding_vector->count; i++) {
        /* Convert binding handle to a string */
        rpc_binding_to_string_binding(binding_vector->binding_h[i],
                                      &string_binding,
                                      &status);    /* error status for this call */
        CHECK_STATUS(status, "Cannot get string binding: ", RESUME);
        printf("%s\n",  string_binding);
        /* Free string binding. */
        rpc_string_free(&string_binding, &_ignore);
  }
#endif           /* TRACING */
  /* Establish entry name value */
  entry_name = (unsigned_char_t *)getenv("RPC_DEFAULT_ENTRY");
```

```
/*
 * Here is were we will handle the elimination of duplicate servers and/or
 * avoid becoming a duplicate server on this node.
 */

/*
 * The following code contacts the directory to determine if this server
 * is currently registered in the name space. This server's host may
 * have crashed, leaving bindings in CDS but no server actually running
 * (and no entry in the endpoint mapper). Or there may be another
 * server for this interface actually running. The code below will
 * distinguish between these two cases and do the appropriate thing: if
 * another server is actually running, this one will exit.
 */
rpc_ns_binding_import_begin(rpc_c_ns_syntax_default,
                            (unsigned_char_t *)entry_name,
                            DCE_lns_v1_0_s_ifspec,      /* interface specification
(DCE_lns.h) */
                            NULL,                       /* No object UUID used */
                            &import_context,
                            &status);          /* error status for this call */
/*
 * If the import was successful then there are bindings in CDS for this
 * interface. So search through the binding handles in the CDS to see
 * if there is binding information for a server on this host.
 */
for (rpc_ns_binding_import_next(import-context, &binding..handle, &status);
   (status == rpc_s_ok) && (match_found == false);
      rpc_ns_binding_import_next(import-context, &binding_handle, &status)) {
      /*
       * With this CDS binding handle loop through and check all this
       * server's bindings handles (all its protocol towers).
       */
      for (i = O; i < binding_vector->count; i++) {
        /*
         * The binding handle returned from the name server matches
         * a binding handle we received from the RPC runtime -- the
         * CDS entry refers to a binding on this host. Now we must
         * check to see if there is a server running on the other
         * end of this binding handle or if it is a stale CDS entry.
         */
```

```
/*      rpc_ep_resolve_binding() --
 *
 * Resolve the partially bound server binding handle into a fully bound
 * server binding handle. This will add the endpoint information for the
 * server to the binding handle. A fully bound server binding handle is
 * required by rpc_mgmt_is_server_listening, called below.
 *
 * The first paramter is the binding handle to resolve. It is an [in,out]
 * paramter to this call. The second parameter identifies the interface
 * whose endpoint is of interest. The final paramter is the DCE error status.
 */
rpc_ep_resolve_binding(binding_handle,
                DCE_lns_v1_0_s_ifspec,/* interface specification (DCE_lns.h) */
                &status);
if (status != rpc_s_ok) {
        /*
         * Then there was something wrong with the binding
         * handle. We cannot reuse it; keep trying.
         */
        CHECK_STATUS(status, "Tried to resolve a binding and got: ", RESUME);
} else {
        /* rpc_mgmt_is_server_listening() --
         *
         * Determine if the server on the other end of this binding handle is
         * listening for remote procedure calls. This should return quickly --
         * i.e., it should not cause an RPC timeout or a DCE exception if there
         * is no server on the other end.
         *
         * The first paramter is a server binding handle for the server of interest.
         * The last paramter is the DCE error status. The routine returns true if
         * there is a server listening or false if no server is there (or if something
         * else went wrong).
         */
        if (rpc_mgmt_is_server_listening(binding_handle, &status) == true) {
          /*
           * We found a valid binding handle! There is a server for this interface
           * currently running on this host. Record this fact and stop checking.
           */
          match_found = true;
          break;
        } else{
          CHECK_STATUS(status, "Checked if server listening and got: ",
                        RESUME);
        }/* else */
```

```
            )/* else binding not resolved */
            /*
             * Make sure to free the binding handle allocated for us by
             * rpc_ns_binding_import_ next() above.
             */
            rpc_binding_free(&binding_handle, &_ignore);
        }/* for all this server's binding handles */
    }/* for all bindings for this interface in CDS */

    /*
     * Close down the association with the name server: free the space
     * allocated for the import context, Ignore the return value.
     */
    rpc_ns_binding_import_done(&import_context, &_ignore);
    if (match_found == true) {
        /*
         * Then a server for this interface is already running on this host,
         * For this application, there should only be one server per host.
         * Print out a message and terminate this server.
         */
        rpc_server_unregister_if(              /* Unregister interface */
                            DCE_lns_v1_0_s_ifspec,     /* interface specification
(DCE_lns.h) */
                            NULL,       /*No object UUID. */
                            &_ignore); /* ignore any errors */
        puts("A LNS server is already running in this cell! Exiting.\n");
        exit(l);
    };
    /*
     * No matching binding handle was found. Do all the work required
     * to register this server with the endpoint mapper and CDS.
     */
    /* rpc_ep_register
     *
     * Register the interface with the local endpoint mapper. This allows connections
     * by applications using this interface without specifying a port (i.e., using a
     * partially-bound binding handle).
     *
     */
    strcpy(annotation, "Logical Name Server");
    rpc_ep_register(                    /* register endpoints in local endpoint map */
      DCE_lns_v1_0_s_ifspec,            /* interface specification (DCE_lns.h)      */
      binding_vector,                   /* the set of server binding handles        */
      NULL,                             /* No object UUID                           */
      (unsigned_char_t *annotation,     /* Annotation for these binding vectors     */
      &status
    );
    CHECK_ STATUS(status, "Can't add address to the endpoint map\n", ABORT);

#ifdef TRACING
    puts("Exporting entry to name service data base (rpc_ns_binding_ export) . ..").
#endif           /* TRACING */
    /* rpc_ns_binding_export() --
     *
     * Export the binding vector and interface specification to the name
     * server. Register in the name service under the host-specific entry
     * name just computed above. The first parameter is the syntax to use;
     * in the first release of DCE there is only one supported syntax. The
     * second parameter is the entry name to look under; it was created
     * above, The third parameter is the server interface specification,
     * with the UUID from the IDL file. The fourth parameter is used to
     * specify an object UUID if the server exports multiple objects; this
     * server does not export multiple objects, so NULL is used.
     *
     */
    rpc_ns_binding_export(              /* export entry to name service database */
      rpc_c_ns_syntax_default,                  /* syntax of the entry name (rpcbase.h) */
      entry_name,                       /* entry name for name service */
      DCE_lns_v1_0_s_ifspec,            /* interface specification (DCE_lns.h) */
      binding_vector,                   /* the set of server binding handles */
      NULL,                             /* No object UUID */
      &status                           /* error status for this call */
    );
    CHECK_STATUS(status,  "Can't export to name service database\n", ABORT);
```

```
 /*
  * Wrap the server listen call with a TRY block to catch any exceptions
  * raised by the RPC server runtime. In addition if an asynchronous
  * signal is received (by the sigwait thread) the listen will be
  * terminated via the rpc_mgmt_ interface.
  */
 TRY {
        puts(''Listcning for remote procedure calls...");
        /*
         * Listen and handle incoming RPC requests. The manager function
         * will be called from the server stub to handle each incoming RPC.
         * The manager must be reentrant since up to max_calls_default
         * threads can be executing that code simultaneously. The listen
         * typically will return only when the appropriate rpc_mgmt_
         * function is called by the sigwait thread below.
         */
        rpc_server_listen(      /* listen for remote calls */
          1,                    /*concurrent calls to server (rpcbase.h)*/
          &status
        );
        CHECK_STATUS(status, "Listen returned with error: %s\n", RESUME);
        puts("Stopped listening...\n");

 } FINALLY{
        /*
         * Remove this server from the namespace, including from any profile
         * or groups it's registered in. Also unexport the bindings and
         * unregister the endpoints with the RPC runtime.
         *
         * NOTE: Not all servers will want to unregister from the name
         * service. If the server is expected to come up again right away
         * it makes more sense to leave the server entries in CDS. However
         * if the server is only running now and again the entry should be
         * removed so clients do not try to contact a server that is no
         * longer listening for requests.
         */
#ifdef TRACING
        puts("Unregistering from NSI...\n");
#endif          /* TRACING */
```

```
        /*
         * Unregister this service from the namespace.
         */
        rpc_ns_binding_unexport(rpc_c_ns_syntax_default, /* default syntax */
                        entry_name,
                                DCE_lns_v1_0_s_ifspec,/* interface specification (DCE_lns.h) */
                        NULL,           /* No object UUID */
                        &_ignore);      /* ignore any errors */
#ifdef TRACING
     puts( "Unregistering  endpoints  and  interface...\n");
#endif          /* TRACING */
        /*
         * Unregister the interface and endpoints with the RPC runtime.
         */
        rpc_ep_unregister(
                        DCE_lns_v1_0_s_ifspec,/* interface specification (DCE_lns.h) */
                    binding_vector,     I* this server's bindings *1
                    NULL,               /*no object UUIDs supported */
                    &status);           /* ignore any errors */
        CHECK_STATUS(status,  "Endpoint unregister failed: ", RESUME);

        rpc_server_unregister_if(
                                DCE_lns_v1_0_s_ifspec,/* interface specification (DCE_lns.h) */
                        NULL,           /* No object UUID. */
                        &status); /*  ignore  any  errors  */
        CHECK_STATUS(status, "Interface unregister failed: ", RESUME);

        /*  rpc_binding_vector_free
         *
         * We are done with the binding_vector so we can free the space.
         */
        rpc_binding_vector_free(        /* free set of server binding handles */
          &binding_vector,
          &status
        );
        CHECK_STATUS(status, "Can't free binding handles and vector\n", ABORT);
   }
```

```
    ENDTRY;
    /*
     * We got here either because the server was told to stop listening or
     * an exception was raised. Some manager functions may still be running
     * in separate threads. A robust server should either wait for these
     * threads to complete gracefully or tell them to terminate (cancel).
     */
    exit(0);
}

/*      sigcatch() --
 *
 * Catch and handle asynchronous signals for the server. This function runs
 * in a separate thread. It awaits receipt of an asynchronous signal using
 * the CMA sigwait call. When one of the signals this thread is waiting for
 * is received by the process this thread will be scheduled. It then tells
 * the server to stop listening, causing the RPC runtime to return from the
 * rpc_server_listen() routine once all the currently running RPC have
 * completed. This thread then exits. When rpc_server_listen() returns the
 * server cleans up its entries from the name space and then exits.
 */
pthread_addr_t sigcatch(pthread_addr_t arg)
{
    sigset_t          mask;            /* signal values to wait for*/
    int               signo;           /* actual signal received */
    unsigned32        status;          /* returned by DCE calls */
    error_status_t    _ignore;         /* returned by DCE calls */

#ifdef TRACING
    puts("Entering signal handling thread.");
#endif          /* TRACING */

    /* sigemptyset() --
     *
     * initialize the signal set pointed to by the first parameter. When
     * initialized the mask includes no signals. Use sigaddset() below to
     * add individual signals to the mask. The mask is used to tell
     * sigwait() which signals to wait for. Any other signals will be
     * ignored.
     */
#ifdef TRACING
    puts("Calling sigemptyset.");
#endif          /* TRACING */
```

```
  if(sigemptyset(&mask)<0)
        perror("sigemptyset failed");

  /* sigaddset() --
   *
   * Add a signal value to a signal mask. The first parameter is the mask
   * which should have been initialized at some point. The second
   * parameter is a signal number which is to be added to the signal mask.
   * The mask parameter is modified to include the signal and returned,
   */
  if(sigaddset(&mask, SIGHUP)<0)
        perror("sigaddset 1 failed");
  if(sigaddset(&mask, SIGINT)<0)
        perror("sigaddset 2 failed");
  if(sigaddset(&mask, SIGTERM)<0)
        perror("sigaddset 3 failed");
#ifdef _POSIX_SOURCE
  /*
   * POSIX defines the following user-defined signals. They are also
   * listed as process-terminating asynchronous signals, so make sure to
   * catch them. There are other process-terminating signals your
   * application may need to catch as well, including SIGALRM, SIGPROF,
   * SIGDIL, SIGLOST.
   *
   * The asynchronous non-terminating signals SIGCONT, SIGPWR and SIGWINDOW
   * can also be caught if desired, but should not cause server process
   * termination.
   */
  if(sigaddset(&mask, SIGUSR1)<0)
        perror("sigaddset 4 failed");
  if(sigaddset(&mask, SIGUSR2)<0)
        perror("sigaddset 5 failed");
#endif /* _POSIX_SOURCE */
```

```
    /* sigwait() --
     *
     * Wait for the receipt of a signal (block this thread). The first
     * argument is the signal mask created above. Only those signal values
     * included in the mask will be waited for. Any other signals will be
     * ignored (will cause process termination or whatever their behavior is
     * defined to be).
     *
     * If no threads were sigwait()ing for the asynchronous signals defined
     * in the mask above and such' a signal were received, the process would
     * die immediately without giving the server a chance to unregister its
     * bindings with the endpoint mapper. Using sigwait() is the only way
     * to catch these asynchronous signals and have the opportunity to clean
     * up before exiting.
     */
#ifdef TRACING
    puts("waiting for a signal,");
#endif          /* TRACING */
    signo = sigwait(&mask);
    printf("Signal %d received! Cleaning up...\n", signo);

    /* rpc_mgmt_stop_server_listening() --
     *
     * Stop the server from listening for more RPC requests. The first
     * parameter is a binding handle indicating the server which should stop
     * listening; a NULL value for this parameter means to stop this server
     * from listening. The final parameter is the DCE error status.
     *
     * This call causes the server runtime to exit from rpc_server_listen()
     * after all currently active RPCS run to completion. Note that no more
     * RPCS will be received once the rpc_server_listen() terminates. If
     * any currently active RPCS don't complete in a timely manner, another
     * signal will kill the server since we will no longer have a thread to
     * catch asynchronous signals!
     */
    rpc_mgmt_stop_server_listening(NULL, &status);
#ifdef TRACING
    puts("checking return code.\n");
#endif          /* TRACING */
    CHECK_STATUS(status, "rpc_mgmt_stop_server error:", RESUME);
}
```

Makefile.hp

```
# FILE NAME: Makefile.hp
#
#HP Makefile for the DCE implementation of LNS
#
# definitions for this make file
#
MACH                    = hp

DEBUG                   = -g
INCENV                  = -1. -I.. -I/usr/include/reentrant
ANSI_FLAGS              = -Aa -D_POSIX_SOURCE
HP_FLAGS           = -D_REENTRANT -DTRACING

CFLAGS                  = ${ DEBUG} ${ ANSI_FLAGS} ${ HP_FLAGS } ${ INCENV}
LDFLAGS                 = ${ DEBUG } -Wl,-a,archive
LIBS               = -lbb -ldce -lm -lc_r

PROGRAMS           = $(MACH)/DCE_lns_server \
                    $(MACH)/register_client \
                    $(MACH)/print_client \
                    $(MACH)/resolve_client \
                    $(MACH)/delete_client
server-. OFILES         = DCE_lns_sstub.o DCE_lns_server.o

IDL_SOURCE              = DCE_lns.idl
HEADERS                 = DCE_lns.h
C.. SOURCE         = DCE_lns_sstub.c DCE_lns_cstub.c
DCE_OBJECTS             = DCE_lns_sstub.o DCE_lns__cstub.o
DCE_PROCEDURES                  = register-.logical_name.o \
                    resolve_logical_name.o \
                    print_logical_names.o \
                    delete_logical_name.o

DCE_PROCEDURES_SOURCE       = register_logical_ name.c \
                    print_logical_names.c

IDLCMD                  = idl -v -cc_opt -D_TIMESPEC_T_

cc                 = cc
```

```
#
# COMPLETE BUILD of the application
#
all:      local ${ PROGRAMS }

#
# Clean
#
clean:
        rm -f *.o hp/* DCE_lns_sstub.c DCE_lns_cstub.c DCE_lns.h

#
# LOCAL BUILD of the client application to test locally
#
local:  interface  register_client.c register_logical_name.c
          $(CC)  $(CFLAGS) -DLOCAL  -o  $(MACH)/local_register \
                register_client.c \
                register_logical_name.c \
                $(LIBS)

          $(CC) $(CFLAGS) -DLOCAL -o $(MACH)/local_print \
                print_client.c \
                print_logical_names.c \
                $(LIBS)

          $(CC) $(CFLAGS) -DLOCAL -o $(MACH)/local_resolve \
                resolve_client.c \
                resolve_logical_name.c \
                $(LIBS)

          $(CC) $(CFLAGS) -DLOCAL -o $(MACH)/local_delete \
                delete_client.c \
                delete_logical_name.c \
                $(LIBS)

#
# INTERFACE BUILD
#
interface:        $(DCE_OBJECTS)   $(DCE_PROCEDURES)

$(DCE_OBJECTS):  $(IDL_SOURCE)
          $(IDLCMD)  $(IDL_SOURCE)
```

```
$(DCE_PROCEDURES) :     $(DCE_PROCEDURES_SOURCE)
        $(CC) $(CFLAGS) -c register_logical_name.c
        $(CC) $(CFLAGS) -c resolve_logical_name.c
        $(CC) $(CFLAGS) -c delete_logical_name.c
        $(CC) $(CFLAGS) -c print_ logical_names.c

#
# CLIENT BUILDS
#
$(MACH)/register_client:     register_client.o DCE_lns_cstub.o
        $(CC) $(CFLAGS) -o $(MACH)/register_client \
              register_client.o DCE_lns_cstub.o $(LIBS)
$(MACH)/print_client:                print_client.o DCE_lns_cstub.o
        $(CC) $(CFLAGS) -o $(MACH)/print_client \
              print_client.o DCE_lns_cstub.o $(LIBS)
$(MACH)/resolve_client:              resolve_client.o DCE_lns_cstub.o
        $(CC) $(CFLAGS) -o $(MACH)/resolve_client \
              resolve_client.o DCE_lns_cstub.o $(LIBS)
$(MACH)/delete_client:               delete_client.o DCE_lns_cstub.o
        $(CC) $(CFLAGS) -o $(MACH)/delete_.client \
              delete_client.o DCE_lns_cstub.o $(LIBS)

#
# SERVER BUILD
#
$(MACH)/DCE_lns_server: $(server_OFILES)  $(DCE_PROCEDURES)  DCE_lns_server.c
        $(CC) $(CFLAGS) -o $(MACH)/DCE_lns_server DCE_lns_sstub.o \
              register_logical_name.o \
              resolve_logical_name.o \
              delete_logical_name.o \
              print_logical_names.o \
              DCE_lns_server.o  $(LIBS)
```

Makefile.sun

```
# FILE NAME: Makefile.sun
#
# Sun Makefile for the DCE implementation of LNS
#
# definitions for this make file
#
MACH                    = sun

DEBUG                   = -g
INCENV                  = -I. -I.. -I/usr/include/reentrant

DEF1            = -D_XOPEN_SOURCE -D_TIMESPEC_T_
DEF2            = -DNO_EXCEPTION_HANDLING -DTRANSARC
DEF3            = -DTRACING -D_POSIX_SOURCE
CFLAGS                  = -Xa $(DEF1) $(DEF2) $(DEF3) ${ DEBUG} $( INCENV}
LDFLAGS                 =${ DEBUG}
LIBS            = -ldce -lthread -lnsl -lm

PROGRAMS        = $(MACH)/DCE_lns_server \
                  $(MACH)/register_client \
                  $(MACH)/print_client \
                  $(MACH)/resolve_client \
                  $(MACH)/delete_client
server_OFILES           = DCE_lns_sstub.o DCE_lns_server.o

IDL_SOURCE              = DCE_lns.idl
HEADERS                 = DCE_lns.h
C_SOURCE        = DCE_lns_sstub.c DCE. lns_cstub.c
DCE_OBJECTS             = DCE_lns_sstub.o DCE_lns__cstub.o
DCE_PROCEDURES                  = register_logical_name.o \
                  resolve_logical_name.o \
                  print_logical_names.o \
                  delete_logical_name.o

DCE_PROCEDURES_SOURCE           = register_ logical_name.c \
                  print_logical_names.c
IDLCFLAGS       = -cc_opt '-Xa -D_TIMESPEC_T_ -D_XOPEN_SOURCE'
IDLFLAGS        = -v
IDLCMD                  = idl $(IDLFLAGS) $(IDLCFLAGS)

CC              = cc
```

```
#
# COMPLETE BUILD of the application
#
all:    local ${ PROGRAMS }

#
# Clean
#
clean:
        rm -f *.o hp/* DCE_lns_sstub.c DCE_ lns_cstub.c DCE_lns.h

#
# LOCAL BUILD of the client application to test locally
#
local:  interface  register_client.c register_logical_name.c
          $(CC) $(CFLAGS) -DLOCAL -o $(MACH)/local_register \
                register_client.c \
                register_logical_ name.c \
                $(LIBS)

          $(CC) $(CFLAGS) -DLOCAL -o $(MACH)/local_print \
                print_client.c \
                print_logical_names.c \
                $(LIBS)

          $(CC) $(CFLAGS) -DLOCAL -o $(MACH)/local_resolve \
                resolve_client.c \
                resolve_logical_name.c \
                $(LIBS)

          $(CC) $(CFLAGS) -DLOCAL -o $(MACH)/local_delete \
                delete_client.c \
                delete_logical_name.c \
                $(LIBS)

#
# INTERFACE BUILD
#
interface:      $(DCE_OBJECTS)   $(DCE_PROCEDURES)

$(DCE_OBJECTS):   $(IDL_SOURCE)
          $(IDLCMD)  $(IDL_SOURCE)
```

```
$(DCE_PROCEDURES):      $(DCE_PROCEDURES_SOURCE)
        $(CC) $(CFLAGS) -c register_logical_name.c
        $(CC) $(CFLAGS) -c resolve_logical_name.c
        $(CC) $(CFLAGS) -c delete_logical_name.c
        $(CC) $(CFLAGS) -c print_logical_names.c

#
# CLIENT BUILDS
#
$(MACH)/register_client:    register_client.o DCE_lns_cstub.o
          $(CC) $(CFLAGS) -o $(MACH)/register_client \
                register_client.o DCE_lns_cstub.o $(LIBS)
$(MACH)/print_client:               print_client.o DCE_lns_cstub.o
          $(CC) $(CFLAGS) -o $(MACH)/print_client \
                print_client.o DCE_lns_cstub.o $(LIBS)
$(MACH)/resolve_client:             resolve_client.o DCE_lns_cstub.o
          $(CC) $(CFLAGS) -o $(MACH)/resolve_client \
                resolve_client.o DCE_lns_cstub.o $(LIBS)
$(MACH)/delete_client:              delete_client.o DCE_lns_cstub.o
          $(CC) $(CFLAGS) -o $(MACH)/delete_ client \
                delete_client.o DCE_lns_cstub.o $(LIBS)

#
# SERVER BUILD
#
$(MACH)/DCE_lns_server:  $(server_OFILES)  DCE_lns_server.c $(DCE_PROCEDURES)
          $(CC) $(CFLAGS) -o $(MACH)/DCE_lns_server DCE_lns_sstub.o \
                register_logical_name.o \
                resolve_logical_name.o \
                delete_logical_name.o \
                print_logical_names.o        \
                DCE_lns_server.o $(LIBS)
```

delete_client.c

```
/* FILE NAME: delete_client.c */
/* This module deletes a logical name in the logical name server, */

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <dce/dce_error.h>
#ifdef LOCAL
#include <dce/uuid.h>
#endif          /* LOCAL */
#include <stdio.h>
#include "DCE_lns.h"   /* header file created by IDL compiler */

#ifdef LOCAL
#define         max_logical_names 100
struct lnt_entry {
      logical_name_t  user_logical_name;
      uuid_t          logical_name_object;
      int             user_count;
} lnt_table_entry [max_logical_names];
#endif          /*  LOCAL*/

main (int argc, char *argv[])
{

   uuid_t               registered_uuid;
   logical_name_t       logical_name;
   unsigned_char_t      *string_uuid;
   error_status_t       return_status;
   unsigned32           dce_call_status;
#ifdef LOCAL
  int              i;

/* Initialize table. */
  for ( i = O; i <= max_logical_names; i++)
  {
    strncpy ( (char *) &lnt_table_entry [i] .user_logical_name,
           " ", 1);
    uuid_create_nil (
              &lnt_table_entry [i].logical_name_object,
              &dce_call_status );
    lnt_table_entry[i].user_count = O;
  }
#endif          /*  LOCAL*/
   if (argc != 2) {
     fprintf(stderr, "Usage: %s logical_name\n", argv[0]);
     exit(l);
   } else {
     strncpy((char *)logical_name, argv[ 1], sizeof(logical_name));
   }

   delete_logical_name ( logical_name, &return_status );

   printf("Returned status = %i.\n", return_status);

)
```

delete_logical_name.c

```
/* File Name: delete_logical_name.c */
/* Purpose: Delete a logical name entry. */

#include <stdio.h>
#include <dce/dce_error.h>
#include <dce/rpcexc.h>
#include <dce/uuid.h>
#include "DCE_lns.h"
#include "DCE_lns_server.h"

void delete_logical_name ( logical_name, call_ status )
  logical_name_t        logical_name;
  error_status_t        *call_ status;
{
  int                   entry_index;
  unsigned32            status;
  uuid_t                nil_uuid;
  unsigned_char_t       *string_uuid;

  for ( entry_index = O ; entry_index < max_logical_names; entry_index++)
  {

        if ( strcmp (
                lnt_table_entry [entry_index] user_ logical_name,
                logical_name) == O ) {
  /*
   * Report the entry being deleted
   */
#ifdef TRACING
             printf(''Deleting entry #- %i logical_name %s.\n",
                entry_index,
                lnt_table_entry [entry_index].user_logical_ name);
#endif          /* TRACING */
  /*
   * Convert the logical name object to a string
   */
             uuid_to_string (&lnt_table_entry [entry_index].logical_name_object,
                &string_uuid,
                &status);
#ifdef TRACING
             printf("   uuid string= %s.\n", string_ uuid);
#endif          /* TRACING */
```

```
    /*
     * Free memory
     */
            rpc_string_free(&string_uuid, &status);

            strncpy (             /* Clear the entry name*/
                    (char *) &lnt_table_entry[entry_index].user_logical_name,
                    " ", 1);
            uuid_create_nil (     /* Clear the entry uuid */
                    &lnt_table_entry[entry_index].logical_name_object,
                    &status );
            lnt_table_entry[entry_index].user_count = O;

            *call_ status = O;
            return;
          }
    )
#ifdef TRACING
    printf("No matching name to delete for %s.\n", logical_name);
#endif              /* TRACING */
    *call_ status = -1;
    return;
}
```

print_client.c

```
/* FILE NAME: print_client.c */
/* This module causes the server to print out a list of all registered names. */

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <dce/dce_error.h>
#ifdef LOCAL
#include <dce/uuid.h>
#endif          /*  LOCAL*/
#include <stdio.h>
#include "DCE_lns.h"   /* header file created by IDL compiler */

#ifdef LOCAL
#define          max_logical_names 100
struct lnt_entry {
      logical_name_t  user_logical_name;
      uuid_t          logical_name_object;
      int            user_count;
 } lnt_table_entry [max_logical_names];
#endif          /*  LOCAL*/

main (int argc, char *argv[])
{

  unsigned32            dce_call_status;
  logical_name_t        logical_name;
  uuid_t                registered..uuid;
  error_status_t        return_status;
  unsigned_char_t       *string_uuid;
#ifdef LOCAL
  int              i;

/* Initialize table. */
  for ( i = 0; i <= max_logical_names; i++)
  {
          strncpy ( (char *) &lnt_table_entry[ i] .user_logical_name,
                  " ", 1);
          uuid_create_nil (
                  &lnt_table_entry [i].logical_name_object,
                  &dce_call_status );
          lnt_table_entry[i].user_count = O;
  }
#endif          /* LOCAL*/

   print_logical_names ( );
}
```

print_logical_names.c

```
/* File Name: delete_logical_name.c */
/* Purpose: Delete a logical name entry. */

#include <stdio.h>
#include <dce/dce_error.h>
#include <dce/rpcexc.h>
#include <dce/uuid.h>
#include "DCE_lns.h"
#include "DCE_lns_server.h"

void print_logical_names ( )
{

  int                    available_name_slots;
  unsigned32             dce_call_status;
  int                    entry_index;
  unsigned_char_t        *string_uuid;

  available_name_slots = O;
  for ( entry_index = O ; entry_index < max_logical_names; entry_index++ )
  {
        if ( lnt_table_entry [entry_index].user_count == O )
          available_name_slots++;
        else {
          printf("Logical Name Table Entry %i.\n", entry_index );
          printf("    logical_name = %s.\n",
                lnt_table_entry [entry_index].user_logical_name );
          uuid_to_string (/* Translate object uuid to string */
                &lnt_table_entry [entry_index].logical_name_object,
                &string_uuid, &dce_call_status);
          printf("    uuid = %s.\n", string_uuid);
          printf("    user_count = %i.\n",
                lnt_table_entry [entry_index].user_count );
  /*
   * Free memory allocated to string uuid
   */
          rpc_string_free(&string_uuid, &dce_call_status);
        }
  }
  printf ( "Available names slots= %i.\n", available_. name_slots);
}
```

register_client.c

```
/* FILE NAME: register_client.c */
/* This module registers a logical names with the logical name server. */

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <dce/dce_error.h>
#ifdef LOCAL
#include <dce/uuid.h>
#endif          /*  LOCAL*/
#include <stdio.h>
#include "DCE_lns.h"    /* header file created by IDL compiler */

#ifdef LOCAL
#define          max_logical_names 100
struct lnt_entry {
      logical_name_t  user_logical_name;
      uuid_t          logical_name_object;
      int             user_count;
 } lnt_table_entry [max_logical_names];
#endif          /*  LOCAL*/

main (int argc, char *argv[])
{

   unsigned32            dce_call_status;
   logical_name_t        logical_name;
   uuid_t                registered_uuid;
   error_status_t        return_status;
   unsigned_char_t       *string_uuid;
#ifdef LOCAL
  int            i;

/* Initialize table. */
  for ( i = 0; i <= max_logical_names; i++)
  {
        strncpy ( (char *) &lnt_table_entry [i] .user_logical_name,
                " ", 1);
        uuid_create_nil (/* Clear table entry */
                &lnt_table_entry [i].logical_name_object,
                &dce_call_status );
        lnt_table_entry [i].user_count = 0;
  }
#endif          /* LOCAL */
  if (argc != 2) {
        fprintf(stderr, "Usage: %s logical_name\n", argv[0]);
        exit(1);
  } else {
        strncpy((char *)logical_name, argv[ 1 ], sizeof(logical_name));
  }

  registered_uuid = register_logical_name ( logical_ name, &return_status );
  if(return_status == 0) {
        uuid_to_string (&registered_uuid, &string_uuid, &dce_call_status);
        printf("uuid string = %s.\n", string_uuid);
        printf("Returned   s t a t u s =   %i.\n", return_status);
  } else
        if(return_status == -01 )
          puts("Duplicate logical name encountered");
        else
          if(return_status == -02)
                puts("Error creating uuid");
  /*
   * Free memory allocated to string_uuid
   */
  rpc_string_free(&string_uuid, &dce_call_status);
}
```

register_logical_name.c

```
/* File Name: register_logical_name.c              */
/* Purpose: Associate a logical name to an object*/

#include <stdio.h>
#include <dce/dce_error.h>
#include <dce/rpcexc.h>
#include <dce/uuid.h>
#include "DCE_lns.h"
#include "DCE_lns_server.h"

uuid_t register_logical_name ( logical_name,call_status )
   logical_name_t      logical_name;
   error_status_t      *call_status;
{
   int                 new_entry_index;
   static int          last_.entry__address;
   unsigned32                   dce_call_status;
   unsigned_char_t     *string_uuid;
#ifdef TRACING
   puts(''Registering  logical  name");
   printf("   Logical name value: %s\n", logical_name);
#endif                 /* TRACING */

/* Start with a clean slate */
   *call_ status = O;
/* Look for duplicates */
   new_entry_index  =  O;
   while ( strcmp  (  lnt_table_entry  [new_entry_.index] .user_logical_name,
               logical_name) != O &
               new_entry_index < max_logical_names )
   {
      new_.entry_index++;
   }
```

```
  if ( new_entry_index < max_logical_names )
  {
    uuid_t      return_uuid;
#ifdef TRACING
    printf("Duplicate name encountered.\n");
    printf("    name = %s.\n", logical_name);
    printf("    entry = %i.\n", new_entry_index);
#endif          /* TRACING */
    *call_status = -01;
    uuid_create_nil ( &return_uuid,
                      &dce_call_status );
    return ( return_uuid );
  }

/* Look for an empty entry in the table. */

  new_entry_index = 0;

  while ( lnt_table_entry [new_entry_index].user_count != 0 &
        new_entry_index < max_logical_names )
  {
    new_entry_index++;
  }
  if ( new_entry_index < max_logical_names)
  {
/* Once we found an available entry, cram a nil uuid value into it. */
/* This is just for good measure.                                   */
     uuid_create_nil (
        &lnt_table_entry [new_entry_index].logical_name_object,
        &dce_call_status );
/* Copy in the user's logical name. */
     strcpy (
        (char *) lnt_table_entry[ new_entry_index].user_logical_name,
        (char *) logical_name );

     lnt_table_entry [new_entry_index].user_count = 1;

     uuid_create (
        &lnt_table_entry [new_entry_index].logical_name_object,
        &dce_call_status );
        if ( dce_call_status != uuid_s_ok )
        {
          uuid_t return_uuid;

          printf("Error creating uuid.\n");

/* We had an error so we need to clear out the table entry. */
          strncpy ( (char *) lnt_table_entry[new_entry_index].user_logical_n ame,
                " ", 1);
          lnt_table_entry[new_entry_index].user_count =0;
          uuid_create_nil ( &return_ uuid,
                      &dce_call_status );
          *call_status = -02;
          return ( return_uuid );
        }
    }

    printf(''new_entry_index = %i.\n", new_entry_index);

    printf ("Returning uuid for %s.\n", logical_name);

    uuid_to_string (&lnt_table_entry[new_entry_index].logical_name_object,
              &string_uuid,
              &dce_call_status);

    printf("uuid string = %s.\n", string_uuid);

    return ( lnt_table_entry[new_entry_index].logical_name_object );
}
```

resolve_client.c

```
/* FILE NAME: resolve_client.c */
/* This module uses the lns server to resolve the logical name input to the */
/* object id, */

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <dce/dce_error.h>
#ifdef LOCAL
#include <dce/uuid.h>
#endif          /* LOCAL */
#include <stdio.h>
#include "DCE_lns.h"   /* header file created by IDL compiler */

#ifdef  LOCAL .
#define         max_logical_names 100
struct lnt_entry {
      logical_name_t user_logical_name;
      uuid_t         logical_name_object;
      int            user_count;
} lnt_table_entry [max_logical_names];
#endif          /*  LOCAL*/

main (int argc, char *argv[])
{

  unsigned32          dce_call_status;
  logical_name_t      logical_name;
  uuid_t              registered_uuid;
  error_status_t      return_status;
  unsigned_..char_t   *string_uuid;
#ifdef LOCAL
 int                 i;

/* Initialize table. */
  for ( i = O; i <= max_logical_names; i++)
  {
        strncpy ( (char *) &lnt_table_entry [i] .user_logical_name,
                 " ", 1);
        uuid_create_nil (
                 &lnt_table_entry    [i].logical_nanm–object,
                 &dce_call_status );
        lnt_table_entry [i].user_count = O;
  }
#endif            /* LOCAL*/
  if (argc != 2) {
        fprintf(stderr, "Usage: %s logical_name\n", argv[0]);
        exit(l);
  } else{
        strncpy((char *)logical_name, argv[1 ], sizeof(logical_name));
  }

  registered_uuid = resolve_logical_name ( logical_ name, &return_status );

  uuid_to_string (&registered_uuid, &string_uuid, &dce_call_status);
  printf("Logical name= %s.\n", logical_name);
  printf(" resolves to uuid string = %s.\n", string_uuid);
  printf(" Returned status= %i.\n", return_status);
  /*
   * Retrun memory allocated to string_uuid
   */
  rpc_string_free(&string_uuid, &dce_call_status);
}
```

resolve_logical_name.c

```
/* File Name: resolve_logical_name.c */
/* Purpose: Resolve a logical name entry. */

#include <stdio.h>
#include <dce/dce_error.h>
#include <dce/rpcexc.h>
#include <dce/uuid.h>
#include "DCE_lns.h"
#include "DCE_lns_server.h"

uuid_t resolve_logical_name ( logical_name, call_ status )
  logical_name_t        logical_name;
  error_status_t        *call_status;
{
  uuid_t        nil_uuid;
  int           entry_index;
  unsigned32  dce_call_status;

  for ( entry_index = O ; entry_index < max_logical_.names; entry_ index++)
  {

        if ( strcmp (/* Check for a match */
                lnt_table_entry [entry_index].user_logical_name,
                logical_name) == O )
        {
#ifdef TRACING
          printf("Received query for logical name= %s.\n",
                lnt_table_entry [entry_index].user_logical_name);
#endif          /* TRACING */
          *call_status = O;
          return ( lnt_table_entry [entry_index].logical_name_object );
        }
  }
#ifdef TRACING
  printf("No match found for logical name= %s.\n", logical_name);
#endif          /* TRACING */
  *call_status = -1;
  uuid_create_nil ( &nil_uuid, &dce_call_status );
  return ( nil_uuid );
)
```

APPENDIX C
DESCRIPTION OF THE ICE CREAM APPLICATION

Introduction

This appendix describes the Ice Cream application. It discusses how to build and run it, and what administrative setup is required to run it. It is assumed that you already have a properly configured and running DCE on the node(s) on which you run this sample application.

Description

The Ice Cream application allows the vendor and customer to share information about the calories per serving of flavors of ice cream. The vendor uses the server process to define a flavor and specify the number the of calories per serving. The customer users the client process to query the servers as to how many calories per serving a specific flavor has.

This application demonstrates the use of object uuids to identify between different servers offering the same servers. In our case each server is offering the ice cream service for a different flavor. The application uses an ACF file to specify implicit binding. This allows the client process to select the specific instance of the server process to use.

The server process uses the LNS application described in Appendices A and B to assure that only one server is running for each flavor. The LNS server uses an interrupt handler to clean up the LNS table, CDS and end point map should it be terminated.

The client process makes use of the calls rpc_ns_import_binding_{ begin, next and done } to select the correct server.

The ice cream server supports a single function:

calories	returns the number of calories for the flavor

This server function is coded in its own C file.

Administrative setup

The Ice Cream server needs to have the LNS server running in the local cell.

The Ice Cream server creates a CDS entry for the ice_cream_group in directory /. :/subsys/HP/sample-apps. It also adds a member to this group for its flavor. Therefore the permissions for the operator of the server must allow write permission to this directory.

Build Process

This sample comes with two makefiles: Makefile.hp and Makefile.sun. Makefile.hp should be used on HP platforms. It places the executables in the hp subdirectory. Makefile.sun should be used on Sun platforms. It places the executable in the sun subdirectory.

The flag -DTRACING is define by default in the makefiles. Turning this flag off will greatly reduce the number of displays printed by the LNS server.

Running the server and clients

To start the server enter:

```
<MACHTYPE>/server <flavor> <calories>
```

To query the number of calories in a flavor enter:

```
<MACHTYPE>/client <flavor>
```

APPENDIX D
SOURCE AND MAKEFILES LISTINGS FOR THE ICE CREAM APPLICATION

Makefile.hp

```
# FILE NAME: Makefile.hp
#
#HP Makefile for the ice_cream application
#
# definitions for this make file
#
MACH                    = hp

DEBUG     =     -g
INCENV          = -I. -1.. -I/usr/include/reentrant
ANSI_FLAGS       = -Aa -D_POSIX_SOURCE
HP_FLAGS        = -D_REENTRANT -DTRACING

CFLAGS          = ${ DEBUG] ${ ANSI_FLAGS} ${ HP_FLAGS} ${ INCENV}
LDFLAGS          = ${ DEBUG } -Wl,-a,archive
# MIPS ULTRIX: DCE, internationalization, DECnet
LIBS          = -lbb -ldce -lm -It--r

APPL            = ice_cream
IDLCMD                  = id] -v
CC              = cc


#
# COMPLETE BUILD of the application.
#
all:      interface $(MACH)/client $(MACH)/server

#
# Clean
#
clean:
          rm -f *stub.* *.o $(MACH)/* $(APPL).h
```

```
#
# INTERFACE BUILD
#
interface:       $(APPL).h  $(APPL)_cstub.o $(APPL)_sstub.o
$(APPL).h $(APPL)_cstub.o  $(APPL)_sstub.o:      $(APPL).idl
            $(IDLCMD) $(APPL).idl

#
# CLIENT BUILD
#
$(MACH)/client:       client.o $(APPL)_cstub.o
            $(CC) $(CFLAGS) -o $(MACH)/client client.o \
                  $(APPL)_cstub.o ../LNS/DCE_lns_ cstub.o $(LIBS)


client. o:client.c
            $(CC) $(CFLAGS) -c client.c
#
# SERVER BUILD
#
$(MACH)/server:       $(APPL).h server.o calories.o \
   $(APPL)_sstub.o
            $(CC) $(CFLAGS) -o $(MACH)/server server.o \
                    calories.o . ./LNS/DCE_lns_cstub.o \
                    $(APPL)_sstub.o  $(LIBS)

server.o: server.c
            $(CC) $(CFLAGS) -c server.c
```

Makefile.sun

```
# FILE NAME: Makefile.sun
#
# Sun Makefile for the ice_cream application
#
# definitions for this make file
#
MACH                    = sun

APPL            = ice_cream
INCENV                  = -1. -I..
IDLCFLAGS  = -cc_opt '-Xa -D_XOPEN_SOURCE -D_TIMESPEC_T_'
IDLFLAGS  =  -V
IDLCMD                  = idl $(IDLFLAGS) $(IDLCFLAGS)
LIBDCE                  = -ldce                         # OSF\1: DCE libraries
# MIPS ULTRIX: DCE, internationalization, DECnet
LIBS            = $(LIBDCE) -lthread -lnsl -lm
DEF1            = -D_XOPEN_SOURCE -D_TIMESPEC_T_ -DTRANSARC
DEF2            = -DTRACING -D_POSIX_SOURCE
CFLAGS                  = -g -Xa $(DEF1) $(DEF2) $(INCENV)
CC              = cc


#
# COMPLETE BUILD of the application.
#
all:      interface $(MACH)/client $( MACH)/server

#
# Clean
#
clean:
          rm -f *stub.* *.o $(MACH)/* $(APPL).h

#
# INTERFACE BUILD
#
interface:          $(APPL).h $(APPL)_cstub.o  $(APPL)_sstub.o
$(APPL).h $(APPL)_cstub.o  $(APPL)_sstub.o:        $(APPL).idl
            $(IDLCMD) $(APPL).idl
```

```
#
# CLIENT BUILD
#
$(MACH)/client:       client.o $(APPL)_cstub.o
            $(CC) $(CFLAGS) -o $(MACH)/client client.o \
                   ../LNS/DCE_lns_cstub.o \
                   $(APPL)_cstub.o $(LIBS)


client.o:  client.c .
            $(CC) $(CFLAGS) -c client.c
#
# SERVER BUILD
#
$(MACH)/server:       $(APPL).h server.o calories.o \
   $(APPL)_sstub.o
            $(CC) $(CFLAGS) -o $(MACH)/server server.o \
                      calories.o . ./LNS/DCE_lns_cstub.o \
                      $(APPL)_sstub.o  $(LIBS)

server.o: server.c
            $(CC) $(CFLAGS) -c server.c
```

calories.c

```
/* FILE NAME: calories.c */
/* PURPOSE: returns the number of calories per serving for the*/
/*          servers flavor.                                   */

extern calories_per_serving;
int calories ()
{
#ifdef TRACING
  printf(''Incoming call received.\n");
#endif          /* TRACING */
  return(calories_per_serving);
}
```

client.c

```
/* FILE NAME: client.c */
/* Purpose: Client of' the ice_cream application. */

#include <stdio.h>
#include <stdlib.h>
#include "ice_cream.h"
#include "../LNS/DCE_lns.h"
#include "../check_status.h"

#define STRINGLEN SO

main(argc, argv)
  int argc;
  char *argv[];
{

 rpc_binding_handle_t        binding_h;
 long                        calories_per_serving;
 unsigned_char_t             *ep_str;
 char                        entry_name[STRINGLEN];
 logical_name_t              flavor;
 rpc_ns_handle_t             import_context;
 unsigned_char_t             *object_uuid;
 unsigned_char_t             *net;
 unsigned_char_t             *netopt;
 unsigned_char_t             *protseq;
 uuid_t                      registered_uuid;
 error_status_t              return_status;
 unsigned32                  status;
 unsigned_char_t             *string_uuid;
 unsigned_char_t             *string_binding;

 if (argc != 2) {
        fprintf(stderr, "Usage: %s flavor\n", argv[0]);
        exit(l);
 ) else {
        strncpy((char *)flavor, argv[1], sizeof(flavor));
 }
 /*
  * Determine object UUID for the flavor supplied
  */
 registered_uuid = resolve_logical_name ( flavor, &return_status );
 if(return_status != O) {
        puts("Unable to locate flavor in flavor data base.");
        puts("Client  terminated.");
        exit(-1);
 }

 /*
  * Convert the object UUID to a string
  *
  * NOTE: We do not free the string here.  We will use it later to make
  * a positive id of the correct server.
  */
 uuid_to_string (&registered_uuid, &string_uuid, &status);
#ifdef TRACING
 printf("Logical name= %s.\n",  flavor);
 printf(" resolves to uuid string = %s.\n", string_uuid);
 printf("  Returned  status= %i.\n", return_status);
#endif            /* TRACING */

 strcpy(entry_name, "/. :/subsys/HP/sample-apps/ice_cream_");
 strcat(entry_name, argv[ 1 ]);
#ifdef TRACING
 printf("Entry  Name=  %s.\n", entry_name);

 puts ("Calling rpc_ns_binding_import_begin. ");
#endif            /* TRACING */
```

```
/*  rpc_ns_binding_import_begin() --
 *
 * Contact the directory service to determine the location information
 * for the server. This call caches information from the directory
 * entry named into a local database. We can then walk through the
 * cached information in the following loop.
 *
 * The first parameter is the syntax to use; in the first release of DCE
 * there is only one supported syntax (the default), so usc it. The
 * second parameter is the entry name to look under; it was created
 * above. If NULL is passed in as the entry name, the environment
 * variable RPC_DEFAULT_ENTRY will be used as an entry name instead
 * (this is done in the code below). The third parameter is the client
 * interface specification, with the UUID from the IDL file; this is
 * used to identify the server in the name service. The fourth
 * parameter is used to specify an object UUID if the server exports
 * multiple objects; this server does not export multiple objects, so
 * NULL is used, The fifth parameter is returned by the call; it points
 * to an opaque data structure holding context information used by the
 * rpc_ns_binding_import_next() routine as it walks through the
 * information retrieved. The last parameter is the DCE error status.
 */
rpc_ns_binding_import_begin( /* set context to import binding handles */
        rpc_c_ns_syntax_default,       /* use default syntax */
        (unsigned_char_t *)entry_name,        /* begin search with this name */
        ice_cream_v1_0_c_ifspec,      /* interface specification (ice_cream.h) */
        NULL,                                 /*no optional object UUID required */
        &import_context,              /* import context obtained. */
        &status
);
CHECK_STATUS(status, "Can't begin import:", RESUME);
```

```
  while(l) {
  /*     rpc_ns_binding_import_next() --
   *
   * Attempt to import a binding for the server. The first parameter
   * is the context returned by the import_begin call above. The
   * second parameter is a binding handle data structure that will be
   * allocated. This application must free the binding handle after
   * wc arc done with it. The last parameter is the DCE error status.
   *
   * This call will randomly return one of the bindings found in the
   * directory (if there are more than one). It returns the status
   * rpc_s_no_more_bindings when the bindings have been exhausted.
   */
#ifdef TRACING
        puts("Calling rpc_ns_binding_import_next.");
#endif          /* TRACING */
        rpc_ns_binding_import_next(          /* import a binding handle*/
          import_context,       /* context from rpsc_ns_binding_import begin */
          &binding_h,                   /* output binding handle*/
          &status
        );
        if(status != rpc_s_ok) {
          if(status == rpc_s_no_more_bindings) {
                puts("Can't import a binding handle: Out of bindings.");
                puts("Program abort.");
                exit(-1);
           } else{
                CHECK_STATUS(status, "Can't import a binding handle:", RESUME);
                break;
           }
        }
  /*
   * Perform application specific selection of binding handle.
   */
#ifdef TRACING
        puts("Calling rpc_binding_to_string_binding.");
#endif          /* TRACING */
        rpc_binding_to_string_binding (       /* convert binding information */
          binding_h,
          &string_binding,
          &status
        );
        CHECK_ STATUS(status, "Can't get string binding:", RESUME);
```

```
#ifdef TRACING
        puts("Calling rpc_string_binding_ parse.");
#endif          /* TRACING */
        rpc_string_binding_parse(    /* get components of string binding */
          string_ binding,       /* the string of binding data */
          &object_uuid,          /* an object UUID string is obtained */
          &protseq,              /* a protocol sequence string IS obtained   */
          &net,                  /* a network address string is obtained */
          &ep_str,               /* an endpoint string is obtained */
          &netopt,               /* a network options string is obtained */
          &status
        );
        CHECK_ STATUS(status, "Can't parse string binding:", RESUME);
  /*
   * Free the memory allocated for stuff we do not need.
   */
#ifdef TRACING
        printf("Calling rpc_string_free for string_binding = %s.\n",
                string_binding);
#endif          /* TRACING */
        rpc_string_free(&string_binding, &status);
#ifdef TRACING
        printf(''Object uuid = %s.\n", object_.uuid);

        printf("Calling rpc_string_free for protseq = %s.\n", protseq);
#endif          /* TRACING */
        rpc_string_free(&protseq, &status);
#ifdef TRACING
        printf("Calling rpc_string_free for net = %s.\n", net);
#endif          /* TRACING */
        rpc_string_free(&net, &status);
#ifdef TRACING
        printf("Calling rpc_string_free for for ep_str = %s.\n", ep_str);
#endif          /* TRACING */
        rpc_string_free(&ep_str, &status);
#ifdef TRACING
        printf("Calling rpc_string_free for netopt = %s.in", netopt);
#endif          /* TRACING */
        rpc_string_free( &netopt, &status);

        if(strcmp(object_uuid, string_uuid) == O) {
#ifdef TRACING
          puts("Match found freeing object uuid.");
#endif          /* TRACING */
```

```
            rpc_string_free(&object_uuid, &status);
#ifdef TRACING
            puts(''Frecing   string_uuid.");
#endif          /* TRACING */
            rpc_string_free(&string_uuid, &status);
            global_binding_handle = binding_h;
            break;
         }
         else {
#ifdef TRACING
            puts("Match NOT found freeing object uuid.");
#endif          /* TRACING */
            rpc_string_free(&object_uuid, &status);
#ifdef TRACING
            puts("Freeing binding handle.");
#endif          /* TRACING */
            rpc_binding_free( /* free binding information not selected */
                  &binding_h,
                  &status
            );
            CHECK_ STATUS(status, "Can't free binding information:", RESUME);
         }
   )/* end while */
#ifdef TRACING
   puts ("Calling rpc_ns_binding_import_done. ");
#endif          /* TRACING */
   rpc_ns_binding_import_done(        /* done with import context */
         &import_context,       /* obtained from rpc_ns_import_binding_begin */
         &status
   );
   CHECK_ STATUS (status, "rpc_ns_binding-in~port–done failed: ", RESUME);
#ifdef TRACING
   puts("Calling calories per serving.");
#endif          /* TRACING */
   calories_per_serving = calories();
   printf(''Calorics per serving for flavor %s = %i.\n",
                  flavor,
                  cal ories_per_serving);
}
```

ice_cream.acf

```
/* FILE NAME: ice_cream.acf */
/* This acttribute configuration file is used in conjunction with the */
/* associated IDL file (ice_cream.idl) when the IDL compiler is invoked. */
[
implicit_handle(handle_t global_ binding_handle) /* use implicit binding method */
]
interface ice_cream
{
}
```

ice_cream.idl

```
/* FILE NAME: ice_cream.idl */
[                                             /* brackets enclose attributes*/
uuid(a703ab82-ae09-11cd-9d11-080009786a45),   /* universal unique identifier*/
version(1.0)                                  /*version of this interface */
] interface ice_cream                         /* interface name          */
{
  /************************Data Type Declarations ************************/

  /************************ Procedure Declarations ************************/
  long calories ();                           /* Return calories */

}/* end of interface definition*/
```

server.c

```
/* FILE NAME: server.c */
/*Purpose: This program implements the ice_cream interface*/

#ifdef TRANSARC
#include <sys/machsig.h>
#endif          /* TRANSARC */
#include <signal.h>         /* For signal handler*/
#include <pthread.h>        /* POSIX threads facility*/
#include <unistd.h>         /* Standard POSIX defines */
#include <stdlib.h>         /* Standard POSIX defines */
#include <string.h>         /* str*() routines */
#include <stdio.h>
#include <dce/dce_error.h>
#include <dce/rpcexc.h>
#include <dce/uuid.h>
#include <ctype.h>
#include <errno.h>
#include "ice_cream.h"              /* header created by the IDL compiler*/
#include "../LNS/DCE_lns.h"         /* LNS header created by IDL compiler */
#include " ../check_status.h" /* contains the CHECK_STATUS macro */

#define STRINGLEN 50

/*
 * This server is a threaded process. To properly handle user-generated
 * (asynchronous) signals wc spawn a new thread that will use the sigwait()
 * CMA routine to await the receipt of an asychronous signal. When such a
 * signal comes in, the server is made to shut down gracefully.
 */
pthread_addr_t sigcatch(pthread_addr_t arg);

/*
 * Declare string routines
 */
char            *strcpy();
char            *strcat();

int             calories_per_serving; /* supplied as a paramter */
logical_name_t          flavor;         /* flavor name supplied as parameter */
```

```
main (argc, argv)
  int argc;
  char *argv[];
{

  char                    annotation[STRINGLEN];    /* annotation for endpoint map */
  rpc_binding_vector_t *binding_vector;     /* binding handle list (rpcbase.h) */
  unsigned32              dce_call_status;
  char                    entry_name[ STRINGLEN]; /* name service entry name */
  char                    group_name[STRINGLEN]; /* name service group name */
  int                     i;
  uuid_vector_t           obj_uuid_vector;
  rpc_protseq_vector_t *protseq_vector;     /*protocol sequence list(rpcbase.h)*/
  uuid_t                  registered..uuid;         /* object id for our logical name */
  error_status_t          return_status;
  pthread_t               sig_thread;               /* Thread id of signal handler*/
  unsigned32              status;                   /*  error  status  (nbase.h) */
  unsigned_ char_t        *string_binding;
  unsigned_char_t         *string_uuid;

  /********************* VALIDATE INPUT PARMETERS *********************/
  if(argc != 3 ) {
        puts("Usage is server <flavor> <calories_per-serving> .\n");
        puts("Server terminated.\n");
        exit  (-l);
  }
  calories_per_serving = atoi(argv[2]);
  if(calories_per_serving == 0) {
        puts("Usage is server <flavor> <calories_per_serving> .\n");
        puts("<calories_per_serving> must be an integer value.\n");
        printf("   atoi error encountered = %i.\n", errno);
        puts("Server terminated.\n");
        exit (-2);
  }
  /************************ OBTAIN UUID FROM LNS ************************/
  strncpy((char *)flavor, argv[ 1 ], sizeof(flavor));
#ifdef TRACING
  printf("Obtaining uuid for object %s.\n", flavor);
#endif          /* TRACING */
  registered_uuid = register_logical_name ( flavor, &return_status );
  if(return_status != O) {
        puts("Unable to register flavor. -- Possibly a duplicate.\n");
        puts("    Server terminating.");
        exit(- 1 );
  }
  uuid_to_string (&registered_uuid, &string_uuid, &dce_call_status);
#ifdef TRACING
  printf ("flavor %s registered with uuid %s\n", flavor, string_uuid );
  printf(''Returned status= %i.\n", return_status);
#endif          /* TRACING */
  /* initialize uuid_vector for rpc_ep_register call */
  obj_uuid_vector. count = 1;
  obj_uuid_vector. uuid [O] = &registered_uuid;

  /****************** START SIGNAL CATCHING THREAD ******************/
  /* pthread_create() --
   *
   * Create a new thread to perform asynchronous signal handling. The
   * pthread_create call spawns a new thread of execution within this
   * process. The first paramter is the address of the thread information
   * data structure; pthread_create will fill this in as a result
   * paramter. The second paramter is the attributes to be used when
   * creating the new thread; the defaults are fine in this case. The
   * third paramter is the name of the function to call when the new
   * thread has been created; when this function returns the thread will
   * terminate. The fourth paramter is used to pass information to the
   * thread routine; the sigcatch routine does not need any additional
   * information, so a NULL pointer is passed in.
   */
  if (pthread_create(&sig_thread, pthread_attr_default, sigcatch, 0) < 0) {
        /*    perror() --
         *
         * Print an error message using the string passed in and the current
         * value of the global UNIX error value, errno. The pthread_create
         * call will set errno if it fails.
         */
        perror("Cannot start signal catching thread");
  } else{
        /*      pthread_yield() --
         *
         * Force a context switch from this thread to another. In this case
        ‘ * there is only the one other thread, the one just spawned. Yeild
         * here to allow the signal thread to set itself up before resuming
         * with registration of the server.
         *
         * NOTE: This does not guarantee that the signal catching thread
         * will run until it blocks (until the sigwait() call, see below).
         */
        pthread_yield();
  }

  /**** ********************* REGISTER INTERFACE ***** *********************/
  /*   rpc_server_register_if
   *
   * Register the interface definition and manager entry point vector with
   * the RPC runtime. This application does not usc type UUIDs (an
   * advanced feature) so specify a nil manager type UUID.
   */
#ifdef TRACING
  puts(’’Registering interface (rpc_server_register_ if) . ..”).
#endif          /* TRACING */
  rpc_server_register_if(
        ice_cream_v 1 _0_s_ifspec,     /* interface specification (ice_cream.h) */
        NULL,                          /*No type uuid */
        NULL,                          /* Use defualt end point manager */
        &status
  );
  CHECK_STATUS(status, “Can’t register interface:", ABORT);
```

```
/**************** CREATING SERVER BINDING INFORMATION ****************/
/* rpc_server_use_all_protseqs() --
 *
 * Specify that the RPC runtime should use all protocol sequences for
 * this application (both UDP/IP and TCP/IP are currently supported).
 * This allows the client the flexibility of choosing whichever protocol
 * sequence it prefers; it also uses more system resources on the server
 * but that's OK for purposes of demonstration.
 *
 * The first parameter specifies the maximum number of concurrent remote
 * procedure call requests that the server can accept. This server
 * wishes to allow only 1 call at a time. The second parameter is the DCE
 * error status.
 */
#ifdef TRACING
  puts("Requesting all protocol sequences (rpc_server_use_all_protseqs) . ..").
#endif          /* TRACING */
  rpc_server_use_all_protseqs(          /* use all protocol sequences */
                              1,        /* queue size for calls= 1 */
                              &status   /* status returned from ths call */
  );
  CHECK_STATUS(status, "Can't register protocol sequences:", ABORT);


  /* rpc_server_inq_bindings
   *
   * Get the bindings handles. The binding information (binding
   * vector return argument binding_vector) is required for registration
   * with the endpoint mapper and the name service.
   */
  puts("Obtaining server binding information (rpc_server_inq_bindings) . ..").
  rpc_server_inq_bindings( /* get all binding information for server */
                              &binding_vector,
                              &status
  );
  CHECK_ STATUS(status, "Can't get binding information:", ABORT);
```

```
/************************** PRINT BINDINGS *****************************/
#ifdef TRACING
  puts("Bindings:\n");

  for (i = 0; i< binding_vector->count;  i++){
        /* Convert binding handle to a string */
         rpc_binding_to_string_binding( /* convert bindings to a string */
                  binding_vector->binding_h[i],
                  &string_binding,
                  &status
         );
         CHECK_STATUS(status, "Cannot get string binding: ", ABORT);
         printf("  Binding  #%i = %s\n", i, (char *)string_binding);
        /*  Free  string  binding*/
         rpc_string_free(&string_binding, &status);
         CHECK_STATUS(status, "Cannot  free  string  binding:", ABORT);
  }
#endif           /* TRACING */

  /****************************** ADVERTISE SERVER ***************************/

  strcpy(entry_name, "/.:/subsys/HP/sample-apps/ice_cream_");
  strcat(entry_name, argv[ 1 ]);

#ifdef TRACING
  printf("Entry  Name=  %s.\n", entry_name);
  puts("Exporting entry to name service data base (rpc_ns_binding_export) . ..").
#endif           /* TRACING */
```

```
   /*  rpc_ns_binding_export() --
    *
    * Export the binding vector and interface specification to the name
    * server. Register in the name service under the host-specific entry
    * name just computed above. The first parameter is the syntax to use;
    * in the first release of DCE there is only onc supported syntax. The
    * second parameter is the entry name to look under; it was created
    * above. The third parameter is the server interface specification,
    * with the U UID from the IDL file. The fourth parameter is used to
    * specify an object UUID if the server exports multiple objects; this
    * server dots not export multiple objects, so NULL is used.
    *
    */
  rpc_ns_binding_export(              /* export to a name service database */
        rpc_c_ns_s yntax_default,      /* syntax of entry name (rpcbase.h) */
        (unsigned_char_t *)entry_name,       /* name of entry in name service */
        icc_crcam_v 1 _0_s_ifspec,     /* interface specification (ice_cream.h) */
        binding_vector,                      /* binding information */
        &obj_uuid_vector,              /* export registered object UUIDs */
        &status
  );
  CHECK_STATUS(status, "Can't export to name service database:", ABORT);

  strcpy(group_name, "/.: /subsys/HP/samplc-apps/ice_crcam_group");
#ifdef TRACING
  printf("Group Name= %s.\n", group_name);
#endif          /* TRACING */
  rpc_ns_group_m br_add(              /* add as member of name service group */
        rpc_c_ns_syntax_default,       /* syntax of group name (rpcbase.h) */
        (unsigned_char_t *)group_name,       /* name of group in name service */
        rpc_c_ns_syntax_default,       /* syntax of member name (rpcbase.h) */
        (unsigned_char_t *)entry_name,       /* name of member in name service */
        &status
  );
  CHECK_STATUS(status, "Can't add member to name service group:", RESUME);

  /**** ********************** MANAGE ENDPOINTS ***** *********************/
  strcpy(annotation, "Icc Cream / ");
  strcat(annotation, argv[ 1 ]);
```

```
 /* rpc_ep_register
  *
  * Register the interface with the local endpoint mapper. This allows
  * connections by applications using this interface without specifying a
  * port (i.e., using a partially-bound binding handle).
  *
  */
rpc_ep_register(                    /* add endpoints to local endpoint map */
      ice_cream_v1_0_s_ifspec,      /* interface specification (ice_cream.h) */
      binding_vector,               /* vector of server binding handles */
      &obj_uuid_vector,             /* export registered object UUIDs */
      (unsigned_char_t *annotation, /* annotation supplied (not required) */
      &status
);
CHECK_STATUS(status, "Can't add endpoints to local endpoint map:", RESUME);

 /* Free binding vector */
rpc_binding_vector_free( /* free server binding handles */
      &binding_vector,
      &status
);
CHECK_STATUS(status, "Can't free server binding handles:", RESUME);

/***************** LISTEN FOR REMOTE PROCEDURE CALLS *****************/
 /*
  * Wrap the server listen call with a TRY block to catch any exceptions
  * raised by the RPC server runtime. In addition if an asynchronous
  * signal is received (by the sigwait thread) the listen will be
  * terminated via the rpc_mgmt_ interface.
  */
 TRY {                                /* thread exception handling macro */
#ifdef TRACING
      puts("Listening for remote procedure calls...");
#endif       /* TRACING */
      rpc_server_listen(    /* Listen for a client's call */
                    1,      /* process one remote procedure call at a time */
                    &status
      );
      CHECK_STATUS(status, "rpc listen failed:", ABORT);

  } FINALLY{
        /*
         * Remove this server from the namespace, including from any profile
         * or groups it's registered in. Also unexport the bindings and
         * unregister the endpoints with the RPC runtime.
         *
         * NOTE: Not all servers will want to unregister from the name
         * service. If the server is expected to come up again right away
         * it makes more sense to leave the server entries in CDS. However
         * if the server is only running now and again the entry should be
         * removed so clients do not try to contact a server that is no
         * longer listening for requests.
         */
#ifdef TRACING
        puts(''Unregistering from NSI...\n");
#endif          /* TRACING */
        /*
         * Unregister this service from the namespace.
         */
        rpc_ns_binding_unexport(
                rpc_c_ns_syntax_ default, /* default syntax */
                (unsigned_char_t *)entry_name,
                ice_cream_v 1 _0_s_ifspec, /* interface specification (DCE_lns.h) */
                &obj_uuid_vector,       /* export registered object UUIDs */
                &status);               /* error return */
        CHECK_STATUS(status, "rpc_ns_binding_unexport error: ", RESUME);

#ifdef TRACING
        puts("Unregistering endpoints and interface...\n");
#endif          /* TRACING */
        /*
         * Unregister the interface and endpoints with the RPC runtime.
         */
       rpc_server_inq_bindings(               /* get binding information */
                &binding_vector,
                &status
       );
       CHECK_ STATUS(status, "Can't get binding information:", RESUME);
       rpc_ep_unregister(
                ice_cream_v 1 _0_s_ifspec, /* interface specification (DCE_lns.h) */
                binding_vector,         /* this server's bindings */
                &obj_uuid_vector,       /* export registered object UUIDs */
                &status                         /* error return */
       );
       CHECK_STATUS(status, "Endpoint unregister failed: ", RESUME);
```

```
        rpc_server_unregister_if(
                ice_cream_v1_0_s_ifspec,/* interface specification (DCE_lns.h) */
                NULL,                       /*No object UUID. */
                &status                     /* error return */
        );
        CHECK_STATUS(status, "Interface unregister failed: ", RESUME);

          /*  rpc_binding_vector_free
           *
           * We are done with the binding_vector so wc can free the space.
           */
        rpc_binding_vector_free(        /* free set of server binding handles */
                &binding_vector,
                &status
        );
        CHECK_STATUS(status, "Can't free binding handles and vector\n", ABORT);
        /*
         * Remove our entry from the logical name service data base
         */
        delete_logical_name ( flavor, &status);
        if ( status != O ) {
          puts("Error deleting flavor name");
        }
  )
  ENDTRY
  /*
   * We got here either because the server was told to stop listening or
   * an exception was raised. Some manager functions may still be running
   * in separate threads. A robust server should either wait for these
   * threads to complete gracefully or tell them to terminate (cancel).
   */
  exit(0);
) /* END SERVER INITIALIZATION */
```

```
/*     sigcatch() --
 *
 * Catch and handle asynchronous signals for the server. This function runs
 * in a separate thread. It awaits receipt of an asynchronous signal using
 * the CMA sigwait call. When one of the signals this thread is waiting for
 * is received by the process this thread will be scheduled. It then tells
 * the server to stop listening, causing the RPC runtime to return from the
 * rpc_server_listen() routine once all the currently running RPC have
 * completed.  This thread then exits. When rpc_server_ listen() returns the
 * server cleans up its entries from the name space and then exits.
 */
pthread_addr_t sigcatch(pthread_addr_t arg)
{
  sigset_t          mask;             /* signal values to wait for */
  int               signo;            /* actual signal received */
  unsigned32        status;           /* returned by DCE calls */

   /* sigemptyset() --
    *
    * Initialize the signal set pointed to by the first parameter. When
    * initialized the mask includes no signals. Use sigaddset() below to
    * add individual signals to the mask. The mask is used to tell
    * sigwait() which signals to wait for. Any other signals will be
    * ignored.
    */
  sigemptyset(&m ask);

   /* sigaddset() --
    *
    * Add a signal value to a signal mask. The first parameter is the mask
    * which should have been initialized at some point. The second
    * parameter is a signal number which is to be added to the signal mask.
    * The mask parameter is modified to include the signal and returned,
    */
  sigaddset(&mask, SIGHUP);
  sigaddset(&mask, SIGINT);
  sigaddset(&mask, SIGTERM);
```

b

```
#ifdef _POSIX_SOURCE
   /*
    * POSIX defines the following user-defined signals. They are also
    * listed as process-terminating asynchronous signals, so make sure to
    * catch them. There are other process-terminating signals your
    * application may need to catch as well, including SIGALRM, SIGPROF,
    * SIGDIL, SIGLOST.
    *
    * The asynchronous non-terminating signals SIGCONT, SIGPWR and SIGWINDOW
    * can also be caught if desired, but should not cause server process
    * termination.
    */
   sigaddset(&mask, SIGUSR1);
   sigaddset(&mask, SIGUSR2);
#endif /* _POSIX_SOURCE */

   /* sigwait() --
    *
    * Wait for the receipt of a signal (block this thread). The first
    * argument is the signal mask created above. Only those signal values
    * included in the mask will be waited for. Any other signals will be
    * ignored (will cause process termination or whatever their behavior is
    * defined to be).
    *
    * If no threads were sigwait()ing for the asynchronous signals defined
    * in the mask above and such a signal were received, the process would
    * die immediately without giving the server a chance to unregister its
    * bindings with the endpoint mapper. Using sigwait() is the only way
    * to catch these asynchronous signals and have the opportunity to clean
    * up before exiting.
    */
   signo = sigwait(&mask);
#ifdef TRACING
   printf("Signal %d received! Cleaning up...\n", signo);
#endif          /* TRACING */
```

```
  /* rpc_mgmt_stop_server_listening() --
   *
   * Stop the server from listening for more RPC requests. The first
   * parameter is a binding handle indicating the server which should stop
   * listening; a NULL value for this parameter means to stop this server
   * from listening. The final parameter is the DCE error status.
   *
   * This call causes the server runtime to exit from rpc_server_listen()
   * after all currently active RPCS run to completion. Note that no more
   * RPCS will be received once the rpc_.server–listeno terminates. If
   * any currently active RPCS don't complete in a timely manner, another
   * signal will kill the server since wc will no longer have a thread to
   * catch asynchronous signals!
   */
  rpc_mgmt_stop_server_listening(NULL, &status);
#ifdef TRACING
 puts(''chccking return code.\n");
#endif          /* TRACING */
 CHECK_STATUS(status, "rpc_mgmt_stop-server error: ") RESUME);
}
```